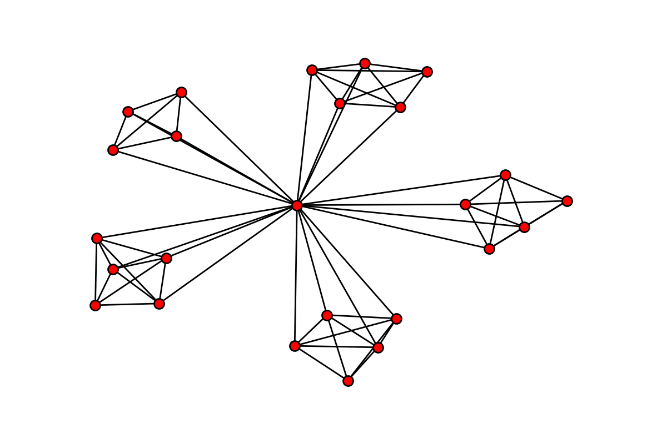
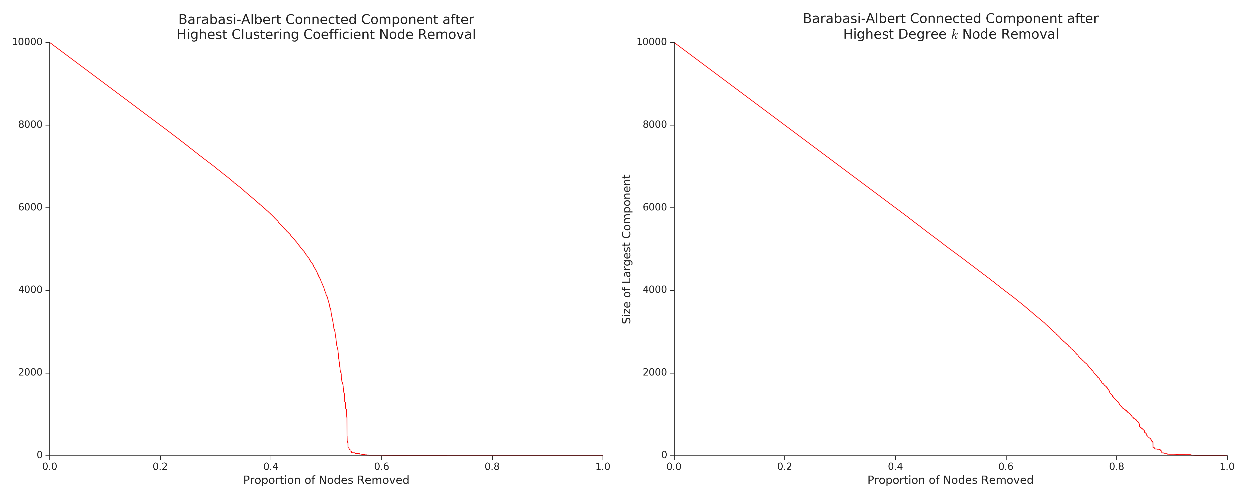
Charles Valentine

Homework 3

11/28/2016 at 5:30 PM

**Problem 1***Code is at end of document*

My computer struggled to perform this exercise and I could not think of a faster way around the computation. I have a cheap CPU and have already logged 7 hours of continuous computation on this problem. I cannot seem to finish the hierarchical model simulation due to an errant crash which does not seem to be caused by anything except the Python kernel itself failing. To prove my implementation of the hierarchical model I have provided this network graphic (1 iteration) showing that I can, at least, generate these networks.



For the Barabási-Albert model of a network the most successful attack strategy to fragment the network into the smallest connected components is to target, explicitly, high clustering coefficient nodes for removal. This makes intuitive sense since nodes with high clustering (and not necessarily high *k*) have a role in joining together clumps of the network. Nodes that are responsible for clustering the graph should be protected more so than nodes with a high degree *k*.

A simulation for random attacks which makes the assumption the attacker does not have any prior information about the role of a target node could be used to asses if all individual’s information should be kept secret. It is likely that, for the Barabási-Albert model the network would be very resilient to random attacks as there are many hubs that provide redundant clustering roles and many, many lowdegree and low clustering nodes that would absorb many attacks with little effect on the network’s structure.

**Problem 2**

1. The rate equation for the time evolution of the degree of node is:
2. Integrating both sides with respect to between and allows us to obtain the degree for node as a function of time:
3. We will use the following inequality to assess the number of nodes with fitness having a degree less than or equal to a given value, .

The number of nodes with fitness having a degree less than or equal to a given value, , is:

1. We obtain the cumulative distribution function of degree for the subset of nodes with fitness by dividing by the total number of nodes of fitness, .

We can then take the derivative with respect to to obtain the associated probability density function.

1. If we assume that the fitness distribution is half nodes with and half nodes the overall degree distribution of the network is:

The average degree of a good looking is the expected value for an exponential distribution. A good-looking node will have so:

Therefore the expected degree of a good looking node is estimated to be . Using the same method we can estimate the expected value for a less good looking node by setting .

The expected degree of a good looking node is estimated to be .

**Problem 3**

1. For the following we discount shuffling of nodes as valid non-degenerate motifs. All six degenerate undirected 4 node motifs (with no unconnected nodes) are summarized in the following table. The number of directed motifs is calculated with only two link types incoming and outgoing directions. However, many of these non-degenerate motifs can be represented by a canonical set. To represent this set we need to remove the motifs that can be represented by a symmetrical mirroring. We can subtract this fraction of patterns by dividing by the number of nodes that lie on a potential symmetry line. There are 38 degenerate directed motifs.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| Nodes | 4 | 4 | 4 | 4 | 4 | 4 |
| Edges | 6 | 5 | 4 | 4 | 3 | 3 |
| Symmetries |  | 2 |  | 1 | 1 | 1 |
|  | 4 | 2 | 4 | 2 | 2 | 4 |
| Non-degenerate Directed Motifs | 64 | 32 | 16 | 16 | 8 | 8 |
| Degenerate Directed Motifs | 16 | 8 | 4 | 4 | 4 | 2 |

1. Estimating how many nodes in an Erdős–Rényi (ER) model graph can be approached by estimating the probability of multiple dependent links between nodes. For an ER graph with nodes and the probability of any two nodes sharing an edge . We first start by estimating how many links exist in the ER graph as and the avg. degree .

Since every node, on average, has 40 links (subtract one from our source node) extending from it we have motifs of three nodes in a row. For extending this model to three nodes we also need to subtract the proportion of links that return from our third node to our source node to keep our model as beads on a string. This subtraction term is represented as which is also the average degree of the node. Using this alternation of expressions depending on the motif structure we were able to estimate the following frequencies.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| Estimated Abundance | 233,99,760 | 231,99,760 | 231,99,920 | 311,999,920 | 234,00,000 | 304,199,920 |

**Code**

%load\_ext autoreload

%autoreload 2

%matplotlib inline

import itertools

import numpy as np

import networkx as nx

import matplotlib as mpl

import matplotlib.pyplot as plt

from operator import itemgetter

mpl.rc('figure', facecolor='white')

mpl.rc('xtick', labelsize=14, color="#222222", direction='out')

mpl.rc('ytick', labelsize=14, color="#222222", direction='out')

mpl.rc('font', size=16)

mpl.rc('xtick.major', size=6, width=1)

mpl.rc('xtick.minor', size=3, width=1)

mpl.rc('ytick.major', size=6, width=1)

mpl.rc('ytick.minor', size=3, width=1)

mpl.rc('axes', linewidth=1, edgecolor="#222222", labelcolor="#222222")

mpl.rc('text', usetex=False, color="#222222")

def cleanup\_chart\_junk(ax):

ax.spines['top'].set\_visible(False)

ax.spines['right'].set\_visible(False)

ax.yaxis.set\_ticks\_position('left')

ax.xaxis.set\_ticks\_position('bottom')

ax.get\_xaxis().tick\_bottom()

ax.get\_yaxis().tick\_left()

return ax

def hierarchical\_graph(i=1):

# Cliques are defined as complete graphs, N=5, with all but one peripheral node.

clique = nx.complete\_graph(5)

for node in clique.nodes():

clique.node[node]['peripheral'] = False if node == 0 else True

# The seed of our graph is a clique.

G = clique.copy()

# Every iteration

for \_ in range(i):

G = nx.disjoint\_union\_all([G, \*itertools.repeat(clique, 4)])

G.add\_edges\_from([(0, node) for node in G.nodes() if G.node[node]['peripheral'] == True ])

clique = G.copy()

return G

nx.draw(hierarchical\_graph(1), pos=nx.spring\_layout(hierarchical\_graph(1)), node\_size=42)

np.random.seed(1)

N\_BA = 1e4

i = 5

BA\_C = nx.barabasi\_albert\_graph(N\_BA, m=10)

BA\_k = BA\_C.copy()

HA\_C = hierarchical\_graph(i)

HA\_k = HA\_C.copy()

N\_BA\_C = []

N\_BA\_k = []

for x in range(1, len(BA\_C)):

# Remove node with highest clustering coefficient

node = sorted(BA\_C.degree().items(), key=itemgetter(1), reverse=True)[0][0]

BA\_C.remove\_node(node)

# Remove node with highest degree k

node = sorted(nx.clustering(BA\_k).items(), key=itemgetter(1), reverse=True)[0][0]

BA\_k.remove\_node(node)

# Record size of largest connected component

N\_BA\_C.append(len(sorted(nx.connected\_components(BA\_C), key=len, reverse=True)[0]))

N\_BA\_k.append(len(sorted(nx.connected\_components(BA\_k), key=len, reverse=True)[0]))

N\_HA\_C = []

N\_HA\_k = []

for x in range(1, len(HA\_k)):

# Remove node with highest clustering coefficient

node = sorted(HA\_C.degree().items(), key=itemgetter(1), reverse=True)[0][0]

HA\_C.remove\_node(node)

# Remove node with highest degree k

node = sorted(nx.clustering(HA\_k).items(), key=itemgetter(1), reverse=True)[0][0]

HA\_k.remove\_node(node)

# Record size of largest connected component

N\_HA\_C.append(len(sorted(nx.connected\_components(HA\_C), key=len, reverse=True)[0]))

N\_HA\_k.append(len(sorted(nx.connected\_components(HA\_k), key=len, reverse=True)[0]))

fig, axes = plt.subplots(2, 2, figsize=(15, 15))

data = [N\_BA\_C, N\_BA\_k, N\_HA\_C, N\_HA\_k,]

labels = ['Barabasi-Albert', 'Barabasi-Albert', 'Hierarchichal', 'Hierarchichal']

styles = itertools.cycle(['Highest Clustering Coefficient', 'Highest Degree $k$'])

sizes = itertools.cycle([N, N\_HA])

for ax, y, label, style, size in zip(axes.flatten(), data, labels, styles, sizes):

ax = cleanup\_chart\_junk(ax)

ax.set\_ylabel('Size of Largest Component')

ax.set\_title(label + ' Connected Component after\n' + style + ' Node Removal')

ax.plot([x / size for x in range(0, len(y))], y, color='red')

for j in [0, 1]:

axes[1, j].set\_xlabel('Proportion of Nodes Removed')

fig.tight\_layout()